

AD-A126 126

TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES  
A DISTRIBUTED GRAPH ALGORITHM: KNOT DETECTION. (U)  
AUG 82 J MISRA, K N CHANDY

F/O 12/1

UNCLASSIFIED

AFOSR-81-0205  
AFOSR-TR-82-0847

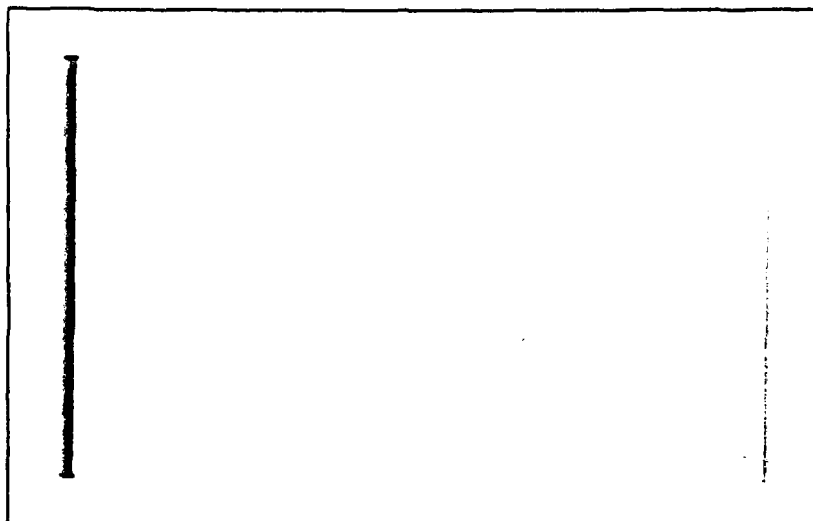
NL

1 of 1  
12/00/82

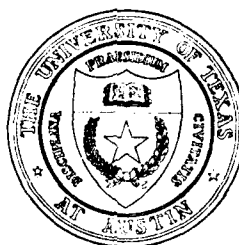



END  
DATE  
FILMED  
11 82  
DTIC

3



AD A120126



**S** **SELECTED**  
OCT 13 1982  
**A**

THE UNIVERSITY OF TEXAS AT AUSTIN  
DEPARTMENT OF COMPUTER SCIENCES

AUSTIN, TEXAS 78712

DTIC FILE COPY

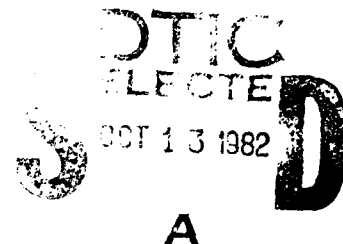
Approved for public release;  
distribution unlimited.

82 10 12 13

A DISTRIBUTED GRAPH ALGORITHM:  
KNOT DETECTION\*

J. Misra  
K. M. Chandy

Department of Computer Sciences  
University of Texas at Austin  
Austin, Texas 78712



ABSTRACT:

A knot in a directed graph is a useful concept in deadlock detection. This paper presents a distributed algorithm based on the work of Dijkstra and Scholten to identify a knot in a graph by using a network of processes.

KEY WORDS AND PHRASES:

Distributed Algorithm, Message Communication, Graph Algorithms.  
Knot

CR Categories: C.2.4, D.1.3, F.2.2, G.2.2

\* This work was supported in part by the Air Force under grants AFOSR 81-0205.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)  
NOTICE OF TRANSMITTAL TO DTIC  
This technical report has been reviewed and is  
approved for public release IAW AFR 190-12.  
Distribution is unlimited.  
MATTHEW J. KERPER  
Chief, Technical Information Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER <b>AFOSR-TR- 82-0847</b>	2. GOVT ACCESSION NO. <b>A120126</b>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) <b>A DISTRIBUTED GRAPH ALGORITHM: KNOT DETECTION</b>	5. TYPE OF REPORT & PERIOD COVERED <b>TECHNICAL</b>	
	6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) <b>J. Misra and K.M. Chandy</b>	8. CONTRACT OR GRANT NUMBER(s) <b>AFOSR-81-0205</b>	
9. PERFORMING ORGANIZATION NAME AND ADDRESS <b>Computer Sciences Department University of Texas Austin TX 78712</b>	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS <b>PE61102F; 2304/A2</b>	
11. CONTROLLING OFFICE NAME AND ADDRESS <b>Directorate of Mathematical &amp; Information Sciences Air Force Office of Scientific Research Bolling AFB DC 20332</b>	12. REPORT DATE <b>August 1982</b>	
	13. NUMBER OF PAGES <b>17</b>	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) <b>UNCLASSIFIED</b>	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) <b>Approved for public release; distribution unlimited.</b>		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES <b>Accepted for publication in <u>ACM Transactions on Programming Languages and Systems</u>.</b>		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) <b>Distributed algorithms; message communication; graph algorithms; knot.</b>		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) <b>A <u>knot</u> in a directed graph is a useful concept in deadlock detection. This paper presents a distributed algorithm based on the work of Dijkstra and Scholten to identify knot in a graph by using a network of processes.</b>		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## 1. INTRODUCTION

A vertex  $v_i$  in a directed graph is in a knot if for every vertex  $v_j$  reachable from  $v_i$ ,  $v_i$  is reachable from  $v_j$ . Chang [1] shows that knot is a useful concept in deadlock detection. Dijkstra [2] has proposed a distributed algorithm for detecting if a given process in a network of processes is in a knot. His algorithm is based on his previous work with C. S. Scholten [3] on termination detection of diffusing computations. We propose an algorithm for knot detection which is also based on [3], but is conceptually simpler. We also discuss the extensions of our algorithm to a more general class of problems.

## 2. MODEL OF A NETWORK OF COMMUNICATING PROCESSES

A process is a sequential program which can communicate with other processes by sending/receiving messages. Two processes  $P$  and  $Q$  are said to be neighbours if they can communicate directly with one another without having messages go through intermediate processes. We assume that communication channels are bi-directional: if  $P$  can send messages to  $Q$  then  $Q$  can send messages to  $P$ . A process knows its neighbours but is otherwise ignorant of the general communication structure of the network.

We assume a very simple protocol for message communication; this protocol is equivalent to the one used by Dijkstra and Scholten [3]. Every process has an input buffer of unbounded

length. If process P sends a message to a neighbour process Q, then the message gets appended at the end of the input buffer of Q after a finite, arbitrary delay. We assume that (1) messages are not lost or altered during transmission, (2) messages sent from P to Q arrive at Q's input buffer in the order sent, and (3) two messages arriving simultaneously at an input buffer are ordered arbitrarily and appended to the buffer. A process receives a message by removing it from its input buffer.

The assumption of unbounded length buffers is for ease of exposition. We show, in section 5.1, that the input buffer length of process Q can be bounded by the number of neighbours of Q.

### 3. A DISTRIBUTED ALGORITHM FOR KNOT DETECTION

Consider a network of processes corresponding to a given directed graph G; there is a one-to-one correspondence between processes in the network and vertices in the graph and a process  $p_i$  in the network represents vertex  $v_i$  in G, for all  $i$ , and  $p_i, p_j$  are neighbours if edge  $(v_i, v_j)$  or  $(v_j, v_i)$  exists in G. Process  $p_1$  initiates a computation to determine if  $v_1$  is in a knot.

#### 3.1 Local Variables of Processes

Every process  $p_i$  maintains the following variables.

**succeeding(i)** : this boolean variable is set true when  $p_i$  determines that  $v_i$  is reachable from  $v_1$ .



Dist		Special	
A			

Initially this variable is false for all  $p_i$ ,  $i \neq 1$  and is true for  $p_1$ . Eventually succeeding(i) will be true if and only if  $v_i$  is reachable from  $v_1$ .

preceding(i) : Same as above except that it represents whether  $v_1$  is reachable from  $v_i$ .

subordinate(i): this is integer valued and will be set to 1 if and only if succeeding(i) and not preceding(i); else it will be set to 0.  $v_1$  is in a knot if and only if subordinate(i) is eventually zero for every process i.

cs(i) : this is an integer valued variable, which keeps the partial sum of some subordinate variables. A goal of the program is to establish the following at termination:

$$cs(1) = \sum_i subordinate(i)$$

Therefore  $v_1$  is in a knot if and only if  $cs(1) = 0$  at termination.

We discuss in section 3.2 the different types of messages sent among processes. In short, a process  $p_i$  may send a message to  $p_j$  and  $p_j$  sends an acknowledgement (ack) to  $p_i$  for every message that  $p_j$  receives from  $p_i$ . We introduce the following variables related to message and ack transmission.

$\text{num}(i)$  : is the number of unacknowledged messages, i.e.  
 the number of messages sent by this process  $p_i$  for  
 which acks have not been received so far.  
 $\text{father}(i)$  : is a process from which  $p_i$ ,  $i \neq 1$ , received a  
 message when its  $\text{num}(i)$  was last zero.  $\text{father}(i)$   
 is undefined initially.

Our goal is to maintain a rooted tree structure at all  
 times over processes whose  $\text{num} > 0$ ;  $\text{father}$  will denote the  
 parent in this tree structure and  $p_1$  the root.

### 3.2 Messages Sent Among Processes

There are two types of messages sent between neighbours  
 in this algorithm.

- (i) Structure message or message : has 2 components  
 (type, p) where, type = suc or pre, and

$p$  is the identity of the sender process. Process  $p_i$  sends  
 (suc,  $p_i$ ) to  $p_j$  if there is a path from  $v_1$  to  $v_j$  in which  $v_i$   
 is the prefinal vertex. Process  $p_i$  sends (pre,  $p_i$ ) to  $p_j$   
 if there is a path from  $v_j$  to  $v_1$  in which  $v_i$  follows  $v_j$  in  
 the path.

- (ii) Acknowledgement message or ack: is of the form (ack, c),  
 where  $c$  is an integer. Acks are used to update  $cs$  and  $\text{num}$ .  
 The entire computation terminates when process  $p_1$  receives  
 acks for all messages that it sent; i.e. when  $\text{num}(1)$  is  
 decremented to zero. Acks for all messages are sent back



as soon as the messages are received except for messages received from father; an ack to a father is sent only when num next becomes zero.

### Convention

It is convenient for purposes of proof to define an atomic action within which invariant assertions may be temporarily violated and outside which the invariants must hold. We write  $\langle A_1; A_2; \dots A_n \rangle$  to show that executions of statements  $A_1, A_2, \dots, A_n$  must be considered as an atomic action. We use Pascal like notation with the added commands send and receive to write our programs.

### 3.3 Knot Detection Algorithm

#### Convention

We write succeeding, preceding, etc. for succeeding(i), preceding(i), when the context is clear.

#### Overview of the Algorithm

As stated earlier, one goal of the algorithm is to maintain a rooted directed tree structure over the set of processes  $p_i$  whose  $\text{num}(i) > 0$ . The root of the tree will be  $p_1$  and  $\text{father}(i)$  will be the parent in the tree for  $p_i$ ,  $i \neq 1$ . In order to maintain the tree structure, we must ensure that, (1) a process  $p_i$ ,  $i \neq 1$ , acquires a father only if it does not have one currently: this is guaranteed since a process acquires a father only when its  $\text{num}(i)$  becomes nonzero, and (2) a process  $p_i$  can be removed from the tree, i.e. set its  $\text{num}(i) = 0$ , only if it was a leaf node:

this will be guaranteed by every process sending its last ack to its father. Computation terminates when the tree is empty.

We will also maintain the invariant (1) given in lemma 4.2, which states that the sum of  $cs$  over all processes plus those in the acks in transit equal the sum of subordinates over all processes. The algorithm will ensure that if  $num(i) = 0$  and  $i \neq 1$ , then  $cs(i) = 0$ . Therefore, when the tree is empty,  $cs(i) = 0$ , for all  $i$ ,  $i \neq 1$  and hence

$$cs(1) = \sum_i subordinate(i).$$

Process  $p_1$  is in a knot if and only if  $cs(1) = 0$ .

### 3.3.1 Algorithm for $p_1$

#### Initialization

```

begin
  father is undefined;
  subordinate := 0; cs := 0; num := 0;
  <succeeding := true;
  num := num + number of successors of  $v_1$ ;
  send(suc,  $p_1$ ) to all successors>;
  <preceding := true;
  num := num + number of predecessors of  $v_1$ ;
  send(pre,  $p_1$ ) to all predecessors>
end

```

#### Upon receiving a structure message (type, p)

send (ack, 0) to p (M1)

#### Upon receiving an acknowledgement (ack, c)

```

begin
  cs := cs + c; num := num - 1; (M2)
  if num = 0 then terminate computation
    { $v_1$  is in a knot if  $cs = 0$ }
end

```

### 3.3.2 Algorithm for $p_i$ , $i \neq 1$

#### Initialization

```

begin
  father is undefined; subordinate := 0; cs := 0, num := 0;
  succeeding := false; preceding := false
end

```

#### Upon receiving a message (type, p)

```

begin
  {update father or send an ack immediately}
  if num = 0
    then father := p
    else begin <send (ack, cs) to p; cs := 0> end;          (L1)

  {update succeeding and preceding if necessary}

  if type = suc and not succeeding {For the first time,  $p_i$ 
    has determined that  $v_i$  is reachable from  $v_1$ }
    then
      begin <succeeding := true;
        num := num + number of successors of  $v_i$ ;
        send (suc,  $p_i$ ) to all successors>
      end;

  if type = pre and not preceding {For the first time,  $p_i$ 
    has determined that  $v_1$  is reachable from  $v_i$ }
    then
      begin <preceding := true;
        num := num + number of predecessors of  $v_i$ ;
        send (pre,  $p_i$ ) to all predecessors>
      end;

  {update subordinate if necessary. Also update cs to maintain
    the invariant in lemma 4.2}
  if succeeding and not preceding
    then
      begin <cs := cs - subordinate + 1; subordinate := 1> end (L2)
    else
      begin <cs := cs - subordinate + 0; subordinate := 0> end; (L3)

  {send ack to father if num = 0}

  if num = 0
    then begin <send (ack, cs) to father; cs := 0> end          (L4)
end

```

Upon Receiving an acknowledgement (ack, c)

```

begin
  cs := cs + c;  num := num - 1;          (L5)
  if num = 0
    then
      begin <send (ack, cs) to father; cs := 0> end (L6)
    end
end

```

#### 4. PROOF OF CORRECTNESS

##### 4.1 Lemma

At any point in the computation, the set of processes with  $\text{num} > 0$  form a rooted tree with  $p_1$  as the root and the parent relation specified by the local variable "father."

##### Proof

The lemma holds vacuously initially.  $\text{num}(i)$  and  $\text{father}(i)$  may be changed only upon receipt of a message or an ack by process  $i$ . If a process with  $\text{num} > 0$  receives a message then it does not alter its father, thus preserving the tree property. Similarly, if a process has  $\text{num} > 0$  after processing an ack, it does not alter the tree structure. If a process  $p_j$  changes  $\text{num}(j)$  from zero then it must have received a message from some other process  $p_i$  on the tree and must have set  $\text{father}(j) = i$ , thus preserving the tree property.

We now show that only a leaf node can decrement its num to zero. If  $p_i$  is on the tree and is not a leaf then there is a process  $p_j$  with  $\text{num}(j) > 0$  and  $\text{father}(j) = i$ ; then  $p_j$  will not return an ack to  $p_i$  while  $p_j$  remains on the tree and hence  $\text{num}(i) > 0$ , while  $p_j$  remains on the tree. Therefore only a leaf node can decrement its num to 0, which preserves the tree property.

Let  $T$ , at any point in computation, denote the set of ack messages which are in Transit, i.e. which have been sent but have not yet been received.

#### 4.2 Lemma

The following is an invariant.

$$\sum_i cs(i) + \sum_{(ack,c) \in T} c = \sum_i subordinate(i) \quad (1)$$

#### Proof

The lemma holds initially since all the terms in the equation are zero. For  $p_i$ ,  $i \neq 1$ , the terms in the equations are modified only at program points L1 through L6, and for  $p_1$ , these terms can be modified only at M1 or M2. The reader may easily convince himself that the equation is left invariant by the execution of the statements at these program points.

#### 4.3 Theorem

Assume that process  $p_1$  terminates computation (in step M2).  $cs(1) = 0$  if and only if  $v_1$  is in a knot.

#### Proof

We will first show that when  $p_1$  terminates computation (I)  $cs(i) = 0$  for  $i \neq 1$ , and (II)  $subordinate(i)$  is correctly set and (III) the set  $T$  is empty. The theorem follows directly from the invariant proven in lemma 4.2.

(I) When  $p_1$  terminates computation in step M2,  $num(1) = 0$ . Then the tree is empty since  $p_1$  was the root of the tree. Therefore  $num(i) = 0$  for all  $i$ . If  $num(i) = 0$  then  $cs(i) = 0$ , for all  $i$ ,  $i \neq 1$ , because every change to  $num(i)$  is followed by the code to set  $cs(i)$  to 0 if  $num(i)$  is 0 (steps L4, L6).

(II) If  $v_i$  is reachable from  $v_1$ , it follows by induction on path length to  $v_i$  that  $p_i$  will eventually receive a message which will result in `succeeding(i)` set true; `succeeding(i)` remains true thereafter. Similarly for `preceding(i)`. Therefore `subordinate(i)` will eventually be set to its correct value. When assignment is made to `succeeding(i)` or `preceding(i)`,  $p_i$  has not returned an ack to its father and hence the computation could not be over. Therefore these variables are assigned their correct values before the termination of computation.

(III) Since the tree is empty, every process must have received acks corresponding to all messages sent. Therefore there can be no ack in transit, i.e. set  $T$  is empty.

#### 4.4 Lemma

$p_1$  will terminate computation in finite time.

#### Proof

A process  $p_i$  sends at most two messages (type,  $p_i$ ), to any other process  $p_j$  because (1) a message is sent only when `succeeding` or `preceding` is set to true and (2) `succeeding` and `preceding` are never reset to false. Because the graph is finite the total number of messages sent is bounded. Hence the total number of acks sent is also bounded. Observe that every process must send or receive either a message or an ack every time it starts to execute. Therefore a process can switch from idle to executing only a finite number of times. There are no loops in the program; therefore every executing process will become idle in finite time. Hence every process in the network will cease to execute in finite time and no more messages or acks will be sent or received from then on.

We now show that the tree must be empty at this point. If not, let  $p_i$  be a leaf node of the tree;  $\text{num}(i) > 0$  since  $p_i$  is on the tree. There is no  $p_j$  on the tree for which  $\text{father}(j) = p_i$  and hence  $p_i$  must have received all its outstanding acks; therefore  $\text{num}(i) = 0$ ! Contradiction!

## 5. NOTES ON THE KNOT DETECTION ALGORITHM

### 5.1 Bounding the Buffer Size

We assumed earlier for purposes of exposition that buffers are of unbounded length. In the knot detection algorithm a process sends at most 2 messages to any neighbour process and therefore no process sends more than 2 acks to any other process. Hence the buffer length for any process need not exceed 4 times the number of neighbours of the process.

### 5.2 Efficiency

This algorithm is superior to the brute-force algorithm in which: (1) process  $p_1$  computes  $\text{successor}^*$ , the set of vertices reachable from  $v_1$  and (2)  $\text{predecessor}^*$ , the set of vertices that can reach  $v_1$  and (3) then declares that  $v_1$  is in a knot if and only if  $\text{successor}^* \subseteq \text{predecessor}^*$ . The computation of  $\text{successor}^*$  ( $\text{predecessor}^*$ ) can be done by using an algorithm similar to the one proposed here - every ack carries with it a set of successors ( $\text{predecessors}$ ). Therefore a successor at distance  $d$  from  $v_1$ , will have its identity transmitted through  $d$  processes to reach  $v_1$ . Total message length will be at least  $O(N^2)$ , for an  $N$ -vertex graph as opposed to  $O(E)$  for our algorithm where  $E$  is the number of edges.

## 6. EXTENSIONS

We show in this section that the ideas in the knot detection algorithm can be extended to solve a very general class of problems. Consider a distributed computation which is initiated by process  $p_1$  sending messages to some of its neighbors. Any other process can send messages only after receiving a message. The computation terminates when no process has any more messages to send and all messages that have been sent have been received. Dijkstra and Scholten [3] were the first to identify this class of computations, which they call diffusing computations. They proposed an algorithm, using the growing and shrinking tree, to detect termination of diffusing computations. Our contribution is to show how the same idea may be exploited to compute a network-wide function of locally computed results.

Let  $\text{local-result}(i)$  denote some computed result at process  $p_i$ , at termination of the entire computation. It is required to compute  $\text{global-result}$  at the termination of computation, where

$$\text{global-result} = f(\text{local-result}(i), \text{ for all } i) \quad (2)$$

where  $f$  is any arbitrary computable function.

The knot detection algorithm computed the global result  $\text{cs}(1)$ ,

$$\text{cs}(1) = \bigcup_i \text{subordinate}(i), \quad (3)$$

$$\text{i.e. } f \equiv \bigcup$$



We propose two schemes to compute network-wide functions. Note that our algorithm can be used to develop distributed algorithms according to the following methodology: in order to compute some global-result, invent a function  $f$  and local-result( $i$ ) satisfying (1) and then design a distributed algorithm to compute local-result( $i$ ) at process  $p_i$ , for all  $i$ . Then superimpose our algorithm to compute the global-result. A variation of this idea appears in [4], where a number of other problems amenable to this approach, are listed.

One difficulty with a straightforward implementation is that a process cannot know when network computation has terminated. Process  $p_i$  knows that network computation can terminate only when  $\text{num}(i) = 0$ ; however,  $p_i$  cannot assert the converse, i.e. that network computation may not have terminated even if  $\text{num}(i) = 0$ . Hence  $p_i$  must send back its current value of  $\text{local-result}(i)$  to its father every time that it decrements  $\text{num}(i)$  to zero. This causes a problem:  $p_i$  may send back a local-result to its father, and subsequently get another message which causes it to compute a new local-result. Therefore  $p_i$  must cancel the old local-result value. We propose two mechanisms for cancelling out-of-date local results: bags and time-stamps.

To simplify exposition in our discussion of cancellation schemes we will assume that there is no delay between sending and receiving a message, i.e. there is never any message in transit: the reader can easily convince himself that the arguments also apply when the transmission delay is not zero.

#### 6.1 Bags

Each process  $p_i$  maintains two bags  $\text{all}(i)$  and  $\text{cancelled}(i)$ . Each bag element is of the form  $(j, \text{local-result}(j))$ . If  $(j, x)$  is an element in  $\text{cancelled}(i)$  then process  $p_j$  has definitely cancelled an out-of-date local-result  $x$ . If  $(j, x)$  is an element of  $\text{all}(i)$ , then at sometime  $p_j$  posted a local

result  $x$ . The elements in  $\text{all}(i)$  are not necessarily current. Every local result that  $p_j$  has posted appears in the union of bags  $\text{all}(i)$ , for every  $i$ . Similarly, all local results that  $p_j$  has cancelled appear in the union of  $\text{cancelled}(i)$ , for every  $i$ . Therefore  $p_j$ 's current local result is in the difference of these two bag unions. In other words, the goal is to maintain the following invariant. Let  $r(j)$  denote the current local result of process  $j$ , and let  $\cup$  denote the union operation over bags.

$$\cup_j (j, r(j)) = \cup_i \text{all}(i) - \cup_i \text{cancelled}(i)$$

Initially,  $\text{all}(i)$  holds the initial local result of  $p_i$  and  $\text{cancelled}(i)$  is empty. To post a current local result  $x$  and cancel the previous local result  $y$ , process  $p_i$  adds  $(i, x)$  to  $\text{all}(i)$  and  $(i, y)$  to  $\text{cancelled}(i)$ .

Two bags abag and cbag are returned with every ack in the form  $(\text{ack}, \text{abag}, \text{cbag})$ . When  $p_j$  sends an ack it takes the elements out of bag  $\text{all}(j)$  and puts them into abag, and similarly puts elements from  $\text{cancelled}(j)$  into cbag, and then sends abag and cbag along with the ack. If  $p_i$  receives  $(\text{ack}, \text{abag}, \text{cbag})$  it adds the contents of abag to  $\text{all}(i)$  and cbag to  $\text{cancelled}(i)$ .

At termination,  $\text{all}(i)$  and  $\text{cancelled}(i)$  will be empty for  $i \neq 1$ , and  $\text{cancelled}(1)$  will contain tuples corresponding to all cancelled local-results, and  $\text{all}(1)$  will contain tuples corresponding to all local-results, current and cancelled. By removing the cancelled results (i.e. elements

of cancelled(1)) from all(1),  $p_1$  can determine the current local-results for all processes. The knot detection algorithm of section 3 uses the bag idea; the information in the two bags have been condensed into a single integer cs. Adding an element (j,x) to all(i) is implemented by incrementing cs(i) by x. Adding an element (j,y) to cancelled(i) is achieved by decrementing cs(i) by y.

#### A Note on Efficiency

The sizes of the bags returned with acks can be reduced by having each process  $p_i$  remove all elements common to all(i) and cancelled(i) from both all(i) and cancelled(i).

#### 6.2 Time-Stamps

Each process  $p_i$  maintains a set  $S(i)$  of triples of the form (j, n(j), local-result(j)) where n(j) is a time-stamp local to process  $p_j$ . When a process  $p_i$  wishes to post a new local-result x (and cancel an out-of-date result) it increments n(i) and adds (i, n(i), x) to S.

When  $p_i$  sends an ack, it sends (ack, S(i)), and then sets S(i) to empty. Upon receiving an ack, (ack, B),  $p_i$  sets S(i) to the union of S(i) and B. Upon termination, S(i) will be empty for all  $i \neq 1$ , and S(1) will contain all tuples (i, n(i), S(i)) that have been sent.  $p_1$  can identify the current local-results because they will be associated with the latest time-stamps.

#### Efficiency

The sizes of the sets returned with acks can be reduced by having each process  $p_i$  discard all elements in S(i) that it can identify as being out-of-date.

### Acknowledgement

We gratefully acknowledge the suggestions of E.W. Dijkstra and C.S. Scholten, on whose work this paper is based. We are also grateful to two unknown referees for their valuable comments.

### References

- [1] Chang, Ernest, "Decentralized Deadlock Detection in Distributed Systems," University of Victoria, Victoria, British Columbia, Canada V8W 2Y2.
- [2] E. W. Dijkstra, "In Reaction to Ernest Chang's Deadlock Detection," EWD702-0, February 21, 1979, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.
- [3] Dijkstra, E. W. and C. S. Scholten, "Termination Detection for Diffusing Computation," Information Processing Letters, Vol. 11, No. 1, pp. 1-4, August 1980, North-Holland Publishing Company.